Universidad Nacional de Asunción
Lic. en Ciencias Informáticas
Facultad Politécnica

Algoritmos de Ordenamiento

Prof. Ing. Derlis Zárate
ProfDerlisZarate@gmail.com

Contenido

- Introducción
- Bubble sort
- Insertion sort
- Selection sort
- Shellsort
- Mergesort
- Quicksort

Introducción

- Qué es la ordenación?
- El problema de la ordenación consiste en ordenar una secuencia de registros de tal forma que los valores de sus claves formen una secuencia no decreciente.
- Esto es, dados los registros $r_1, r_2, \ldots, r_n$, con valores de clave $k_1, k_2, \ldots, k_n$, respectivamente, debemos producir la misma secuencia de registros en orden $r_{i1}, r_{i2}, \ldots, r_{in}$, tal que $k_{i1} \leq k_{i2} \leq \cdots \leq k_{in}$

Introducción

- Algunos algoritmos de ordenación pueden ejecutarse sobre el mismo array a ordenar (in-place), con el uso de al menos $\mathbf{O}(1)$ memoria adicional (ejemplo: número fijo de variables locales).
- Otros algoritmos en cambio requieren el uso de un segundo array de tamaño equivalente al array a ordenar (requiriendo $\mathbf{O}(n)$ memoria adicional).
- Por cuestiones de eficiencia, se prefieren los del primer tipo.

Introducción

- Los diferentes algoritmos de ordenación pueden clasificarse de acuerdo al tipo de operaciones que realizan:



  - Inserción
  - Intercambio
  - Selección
  - Mezcla
  - Distribución

Introducción

- Así como también pueden clasificarse según el esquema de implementación en relación a la forma de acceder al array:
  - Algoritmos Iterativos
  - Algoritmos Recursivos

Introducción

- El tiempo de ejecución de los algoritmos de ordenación que estudiaremos caen en alguna de estas categorías:

$$\mathbf{O}(n \log(n)) \qquad \mathbf{O}(n^2)$$

- También existen algoritmos de ordenación $\mathbf{O}(n)$

- El tiempo de ejecución del algoritmo puede cambiar significativamente en base al escenario considerado.

Introducción

- Si la cantidad de datos a ordenar puede almacenarse y manejarse en la memoria principal de la computadora, decimos que la ordenación realizada es del tipo ***interna***.
- En cambio si la cantidad de datos a ordenar es muy grande y los registros están almacenados en el disco, decimos que la ordenación realizada es del tipo ***externa***.

Bubble Sort

- Repasando un poco, revisaremos el método de ordenación conocido como Bubble Sort (burbuja), el cual es cuadrático.
- Suponiendo que tenemos un array de datos desordenados:
  - Comenzando en el inicio del array, recorremos el mismo, encontramos el mayor elemento y lo movemos a la última posición.
  - En cada iteración subsecuente, encontramos el siguiente elemento mayor y lo movemos hacia “arriba” en el array.

Bubble Sort

- Comenzando con el primer item del array, asumimos que es el mayor elemento.
- Lo comparamos con el segundo item:
  - Si el primero es mayor, los intercambiamos,
  - Sino, asumimos que el segundo item es el mayor
- Continuamos hasta el fin del array ya sea realizando intercambios entre items adyacentes o redefiniendo el item mayor.
- Este proceso se repite n-1 veces.

Bubble Sort

- Existen variantes que mejoran el bubble sort, pero las alternativas siguen siendo cuadráticas.

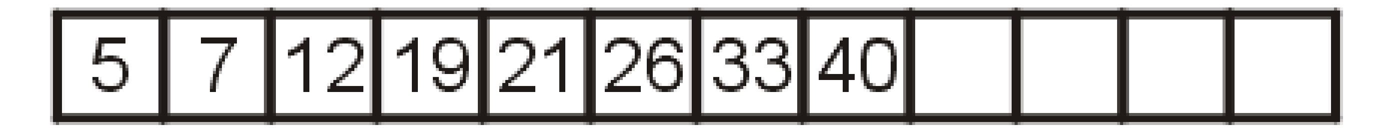
- Ahora veremos varias alternativas mejores.

Insertion Sort

- Consideremos las siguientes observaciones
  - Una lista de 1 elemento está ordenada.
  - En general, si tenemos una lista ordenada de k elementos, podemos insertar un nuevo elemento para crear una lista ordenada de tamaño $k + 1$.

Insertion Sort

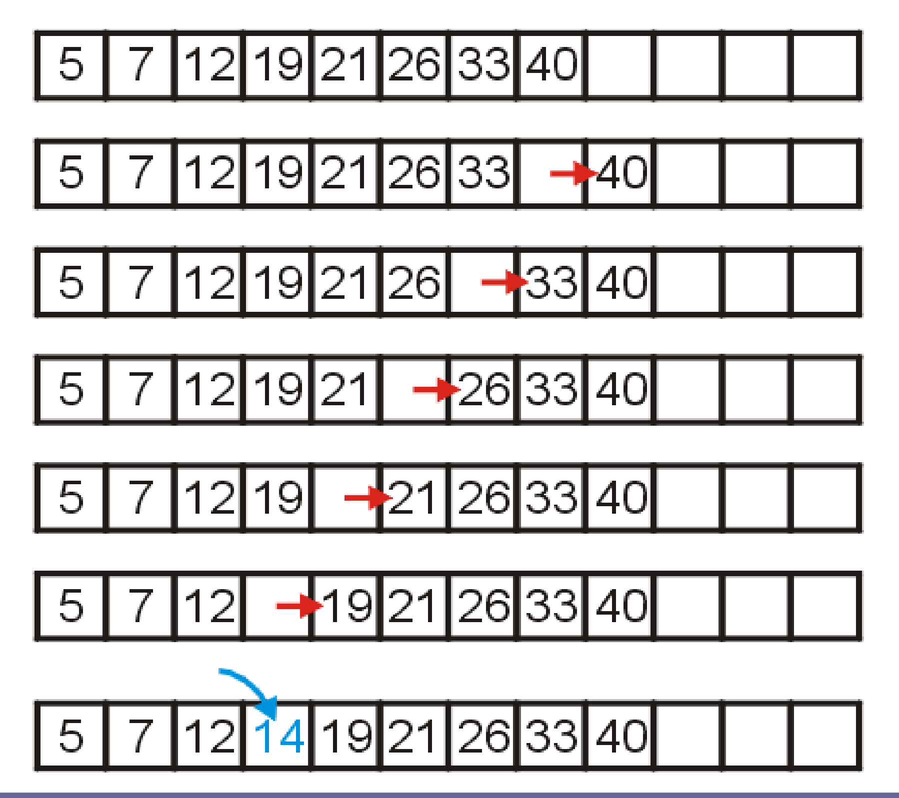
- Por ejemplo, si consideramos el siguiente array ordenado con $k = 8$ entradas



- Queremos insertar el 14 en este array de manera que la lista siga ordenada.

Insertion Sort

- Comenzando al final del array, si el número es mayor que 14, copiarlo a la derecha
- Una vez que se encuentre un elemento menor que 14, insertar el elemento en la vacancia resultante, así:

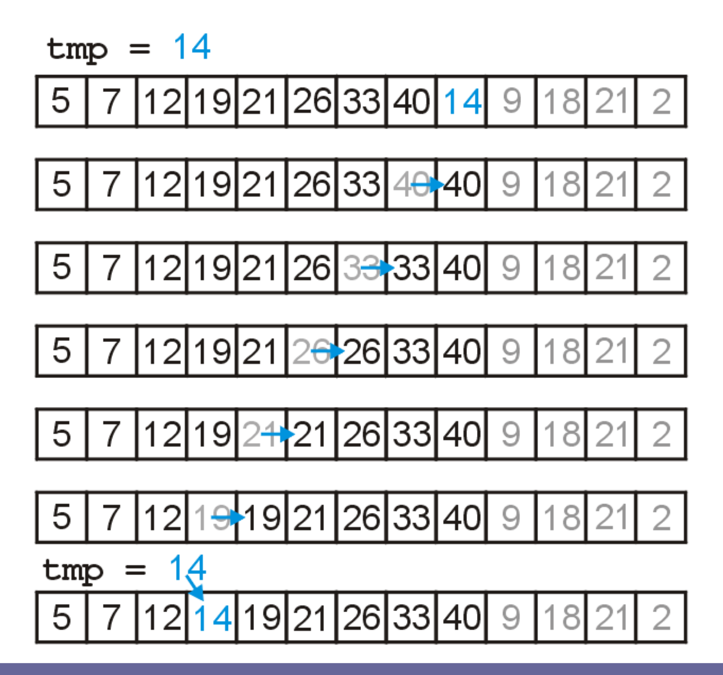


Insertion Sort

- Para cualquier lista desordenada
  - Tratar al primer elemento como una lista ordenada de tamaño $k = 1$
- Luego, dada la lista ordenada de tamaño k
  - Insertar el $(k + 1)^{\text{th}}$ elemento en la lista ordenada
  - La lista ordenada ahora es del tamaño $k + 1$

Insertion Sort

- Para evitar hacer el swap en las asignaciones, podemos almacenar el siguiente elemento en una variable temporal:
  - Esto reduce la asignación en un factor de 3
  - Acelera el algoritmo en un factor de 2



Insertion Sort

- El tiempo de ejecución promedio es

$$\mathbf{O}(\, I + n \,)$$

- Donde I es la cantidad de inversiones del array.
- Una inversión en un array consiste en un par de elementos almacenados en i < j, tal que   a[i] > a[j].
- Cada swap realizado por el algoritmo corrije 1 inversión del array.

Insertion Sort

- Una lista aleatoria puede tener $I = \mathbf{O}(n^2)$ inversiones (por la cantidad de combinaciones posibles)
- Sin embargo, el algoritmo puede ejecutarse en tiempo $\mathbf{O}(n)$ si:
  - Solo un pequeño número de elementos están ubicados fuera de posición, y
  - Los elementos restantes quedarán a pocos lugares de su posición actual.

Insertion Sort

- Beneficios del algoritmo
  - Fácil de implementar
  - Incluso en el peor caso, es rápido para entradas pequeñas

Tamaño	Tiempo Aproximado
64	8000 ns
32	2700 ns
16	750 ns
8	175 ns

Insertion Sort

- Desventajas del algoritmo
  - No es bueno para ordenar grandes entradas de datos.
  - Ordenar una lista aleatoria de tamaño $2^{23} \approx 8\ 000\ 000$ requerirá aproximadamente 1 día.
  - Si doblamos este tamaño de entrada, se cuadruplica el tiempo requerido.

Selection Sort

- Es un algoritmo $O(n^2)$
- Es simple y fácil de implementar, pero ineficiente para listas de tamaño grande.
- En promedio, mejor que el bubble sort pero peor que el insertion sort.

Selection Sort

- Su funcionamiento es el siguiente:
  - Buscar el mínimo elemento de la lista
  - Intercambiarlo con el primer elemento
  - Buscar el mínimo en el resto de la lista
  - Intercambiarlo con el segundo elemento
- Y en general:
  - Buscar el mínimo elemento entre una posición *i* y el final de la lista
  - Intercambiar el mínimo con el elemento de la posición *i*

Selection Sort

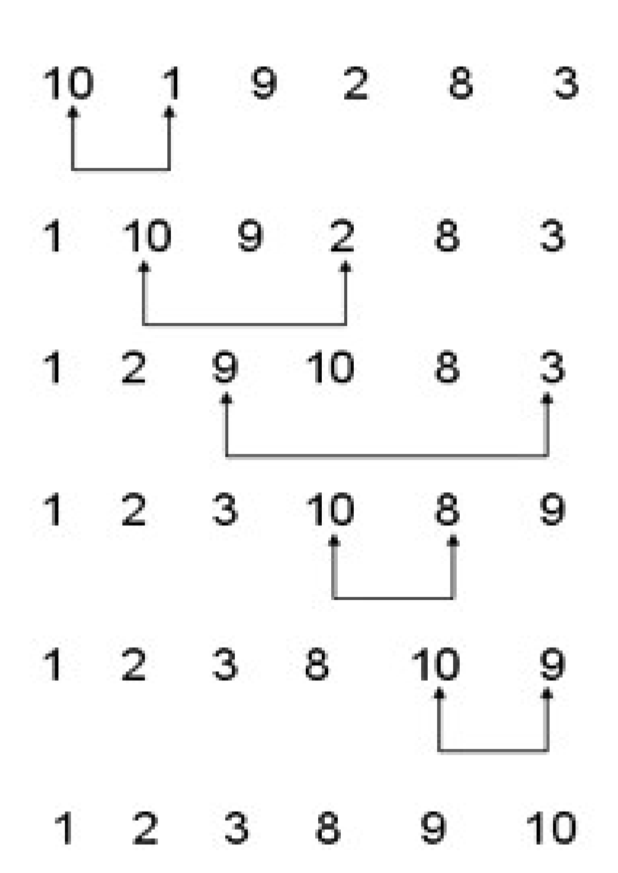
Swap entre el 10 y 1

Swap entre el 10 y 2

Swap entre el 9 y 3

Swap entre el 10 y 8

Swap entre el 10 y 9



Shell Sort

- Propuesto por Donald Shell en 1959.
- Fue el 1er algoritmo en romper la barrera cuadrática, pero recién varios años después se demostró un tiempo sub-cuadrático.
- Shellsort trabaja comparando elementos que están **distantes** en vez de comparar elementos que están adyacentes en el array a ordenar.

Shell Sort

- Utiliza una secuencia $h_1, h_2, \ldots, h_t$ llamada la **secuencia de incremento.**
- Puede utilizarse cualquier secuencia de incremento, con tal que $h_1 = 1$
- Algunas secuencias de incremento dan mejor resultado que otras.

Shell Sort

- El algoritmo realiza múltiples pasadas sobre la lista y ordena conjuntos de igual tamaño utilizando insertion sort.
- Mejora la eficiencia del insertion sort al desplazar rápidamente los valores a sus destinos.

Shell Sort

- El algoritmo también es conocido como ***ordenación por disminución de intervalos***.
- La distancia entre comparaciones va decreciendo mientras el algoritmo se ejecuta, hasta que en la última fase, los elementos adyacentes son comparados.

Shell Sort

- Después de cada fase y cierto incremento h_k, para cada *i,* tenemos que a[***i***] ≤ a [***i*** + h_k]
- Todos los elementos espaciados en una distancia h_k están ordenados.
- Se dice que la lista está *h_k – ordenada*.

Shell Sort

- La ventaja del shell sort es que es eficiente para tamaños de listas moderados.
- Para listas de tamaño grande, el algoritmo no es la mejor opción.
- Es el más rápido de los algoritmos cuadráticos.
- En promedio, aproximadamente 5 veces más rápido que el bubble sort y casi 2 veces más rápido que el insertion sort.

Shell Sort

- La desventaja del shell sort es que su eficiencia no es tan buena como el merge sort o quick sort.
- Por más que sea más lento que dichos algoritmos, es relativamente simple, lo cual lo hace una alternativa válida para ordenar listas de tamaño menor o igual a 5000 elementos.
- También es una buena elección para ordenar listas pequeñas.

Shell Sort

- El tiempo de ejecución del shell sort depende de la selección adecuada de la secuencia de incrementos.
- Existen algunas propuestas conocidas.

Shell Sort

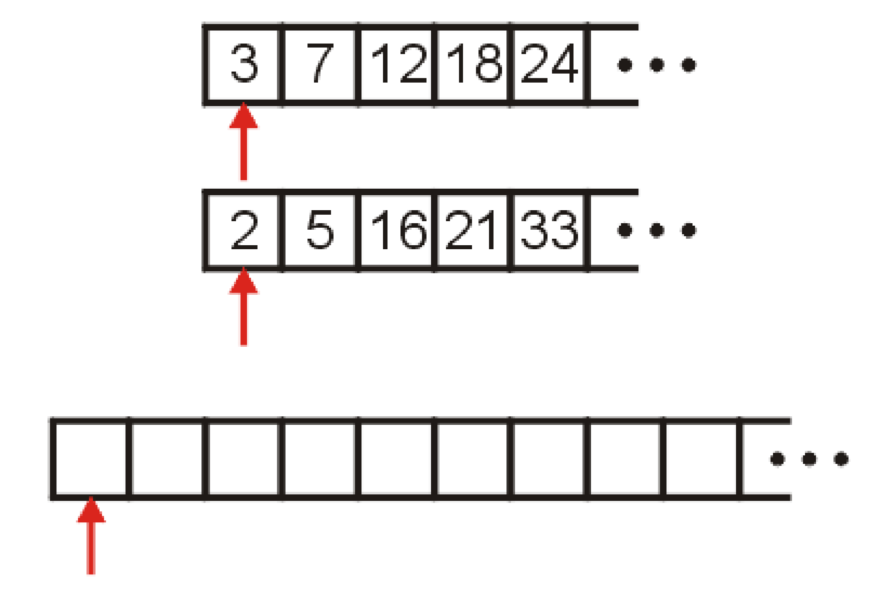
Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

Merge Sort

- Es un algoritmo de ordenación que se define recursivamente.
- No tiene caso peor ni caso mejor, en todos los casos es O (n log n)
- Suponga que:
  - Se divide una lista desordenada en 2 sublistas y,
  - Se ordena cada una de las sublistas
- Qué tan rápido podemos recombinar las 2 sublistas en una única lista ordenada?

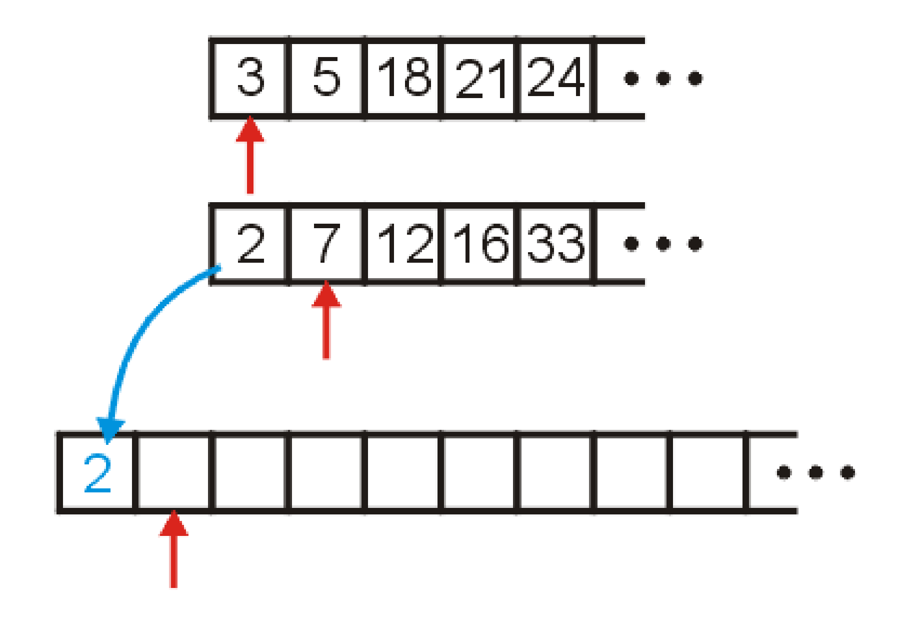
Merge Sort

- Considere 2 arrays ordenados y 1 array vacío
- Definamos 3 índices al inicio de cada array



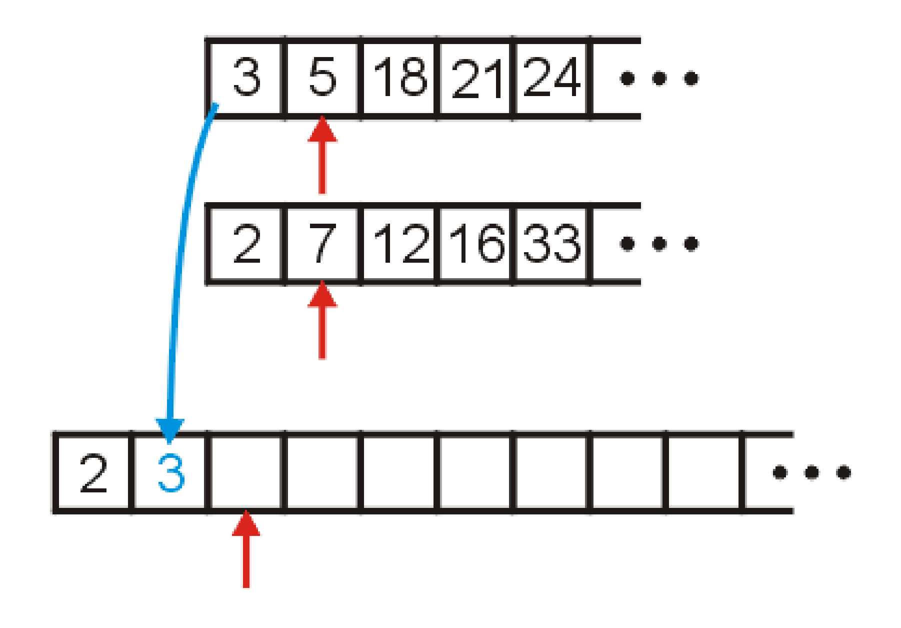
Merge Sort

- Comparamos el 2 y el 3:   2 < 3
- Copiamos el 2 abajo
- Incrementamos los índices correspondientes



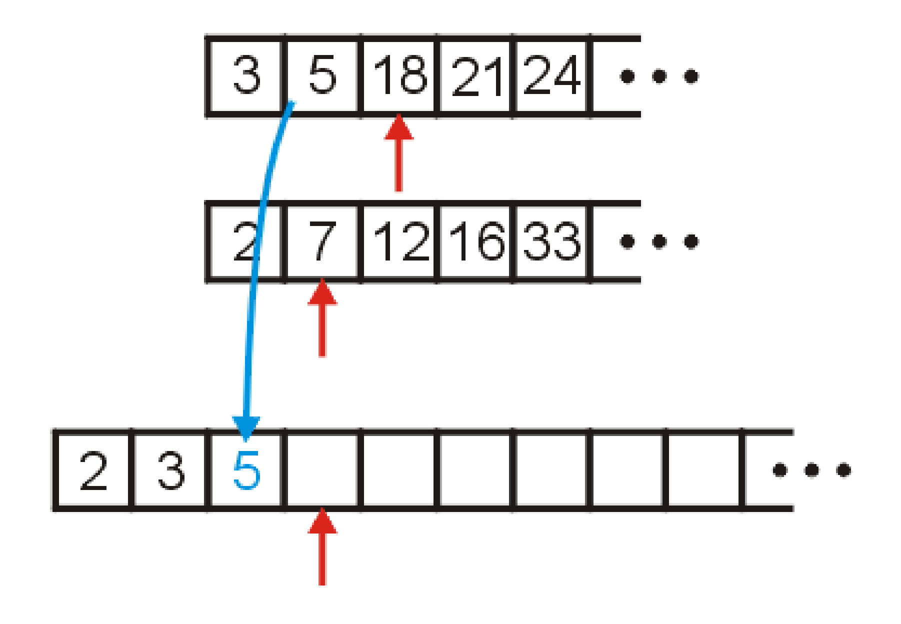
Merge Sort

- Comparamos 3 y 7
- Copiamos el 3 abajo
- Incrementamos los índices



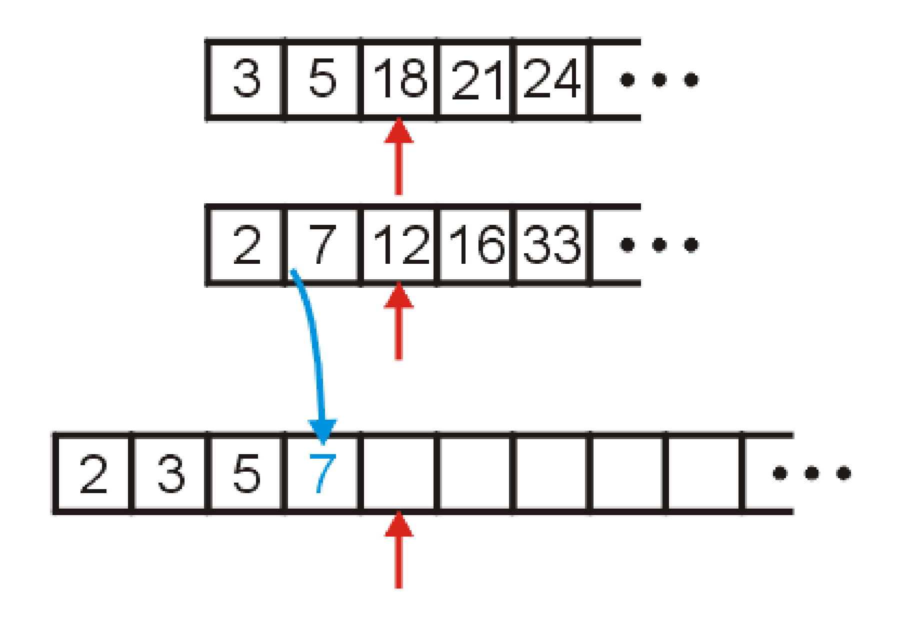
Merge Sort

- Comparamos 5 y 7
- Copiamos el 5 abajo
- Incrementamos los índices



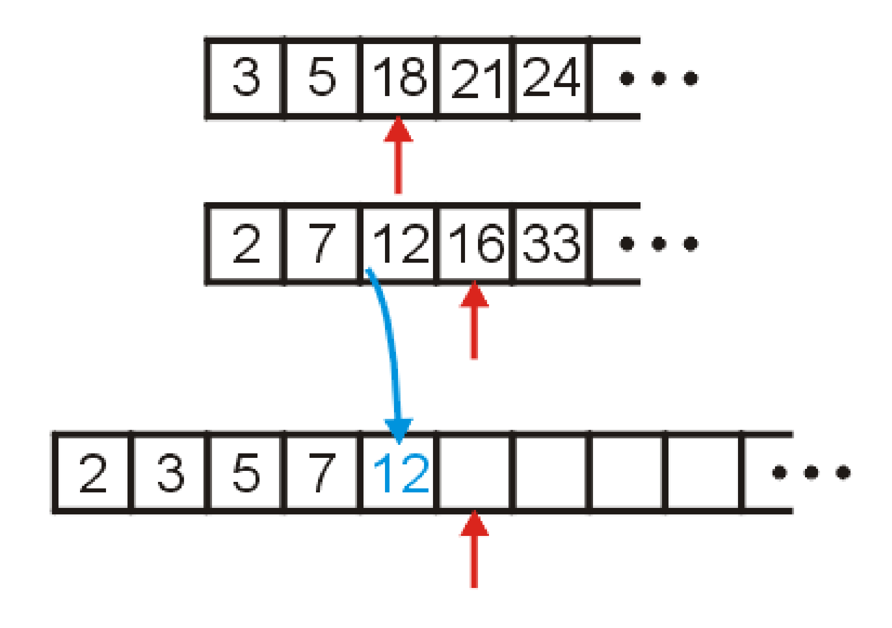
Merge Sort

- Comparamos 7 y 18
- Copiamos el 7 e incrementamos índices



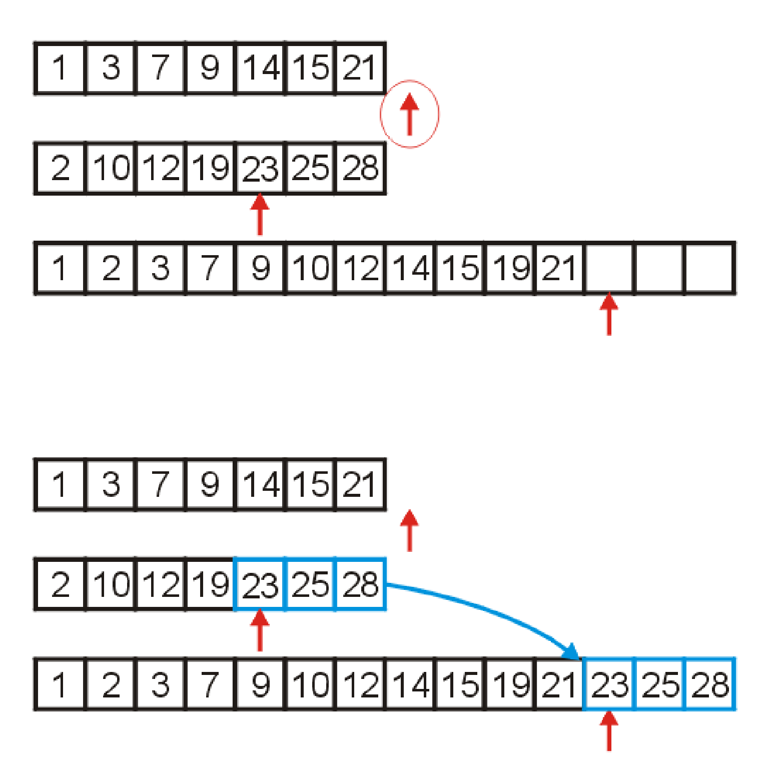
Merge Sort

- Comparamos 12 y 18
- Copiamos el 12 e incrementamos



Merge Sort

- Podemos continuar así hasta terminar de recorrer 1 de los arrays y en ese caso, simplemente copiamos los elementos restantes del otro array, al array vacío



Merge Sort

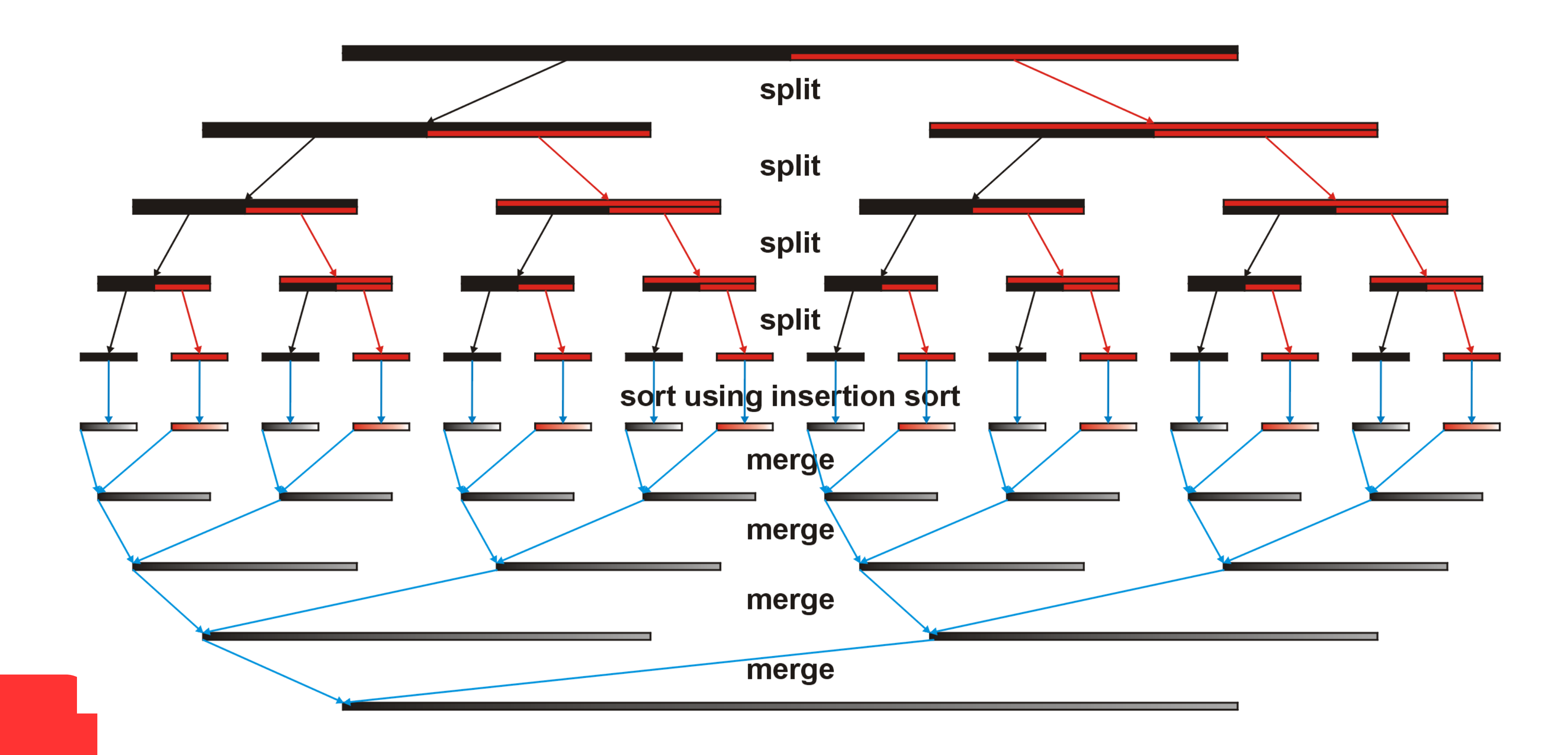
- Entonces, el algoritmo de mezcla puede ejecutarse en tiempo **O**(n)
- Pregunta:
  – Partimos la lista en 2 sublistas y las ordenamos
  – Pero ¿cómo ordenamos las 2 sublistas?
- Respuesta (teórica):
  – Si el tamaño de estas sublistas es > 1, usar merge sort de nuevo.
  – Si el tamaño de las sublistas es 1, no hacer nada, una lista de 1 elemento está ordenada.

Merge Sort

- Sin embargo, el hecho que el algoritmo tenga excelentes propiedades asintóticas no significa que es aplicable a todos los niveles.
- Por ello, la respuesta práctica a la pregunta anterior es:
  - Si el tamaño de las sublistas son menores a un límite establecido, usar un algoritmo como el de insertion sort para ordenar los elementos.
  - De lo contrario, usar merge sort de nuevo.

Merge Sort

- Una interpretación gráfica del merge sort podría ser:

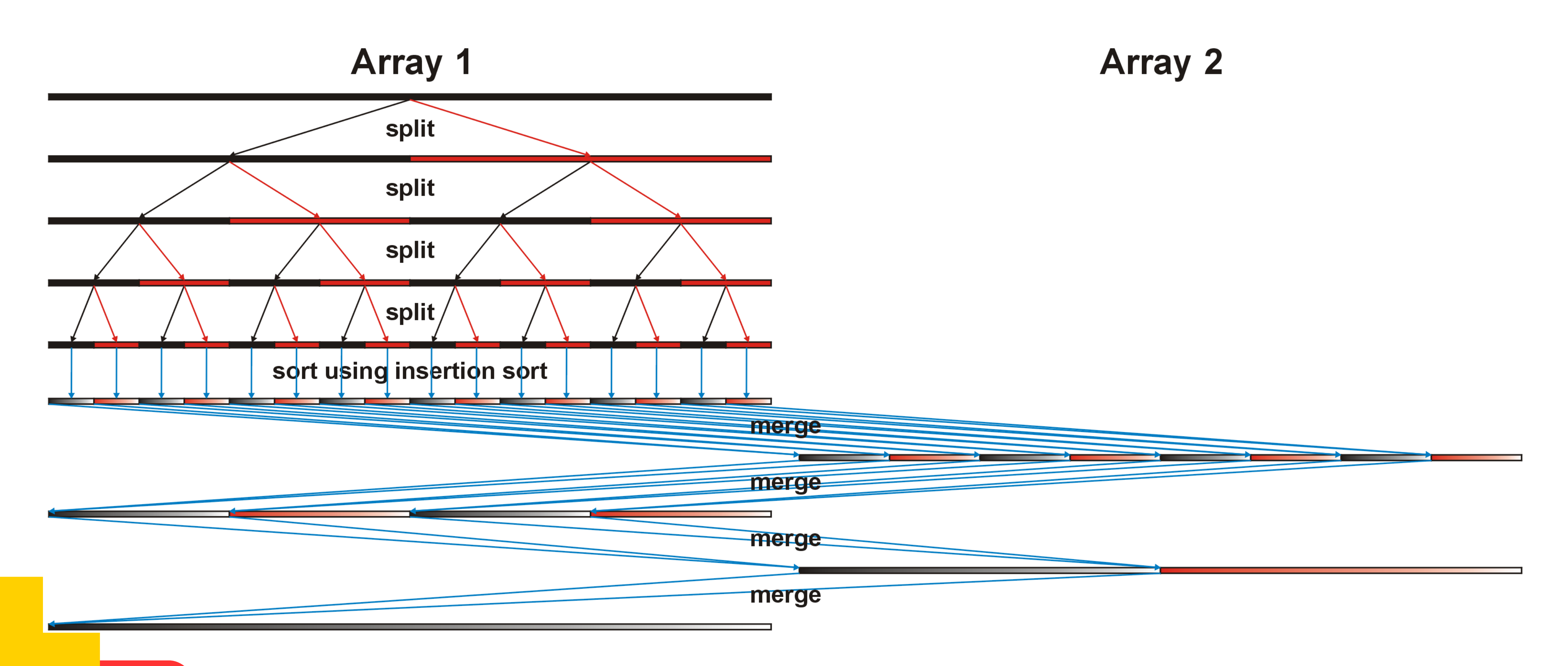


Merge Sort

- Detalles importantes del algoritmo:
  - Si la lista es impar, se divide en 2 sublistas de tamaño aproximado, una par y otra impar.
  - Cada mezcla requiere un array adicional
    - Aunque puede minimizarse el uso de la memoria requerida utilizando 2 arrays, partiendo y ordenando en uno de ellos y mezclando los resultados entre los 2 arrays.

Merge Sort

- Interpretación gráfica usando 2 arrays



Merge Sort

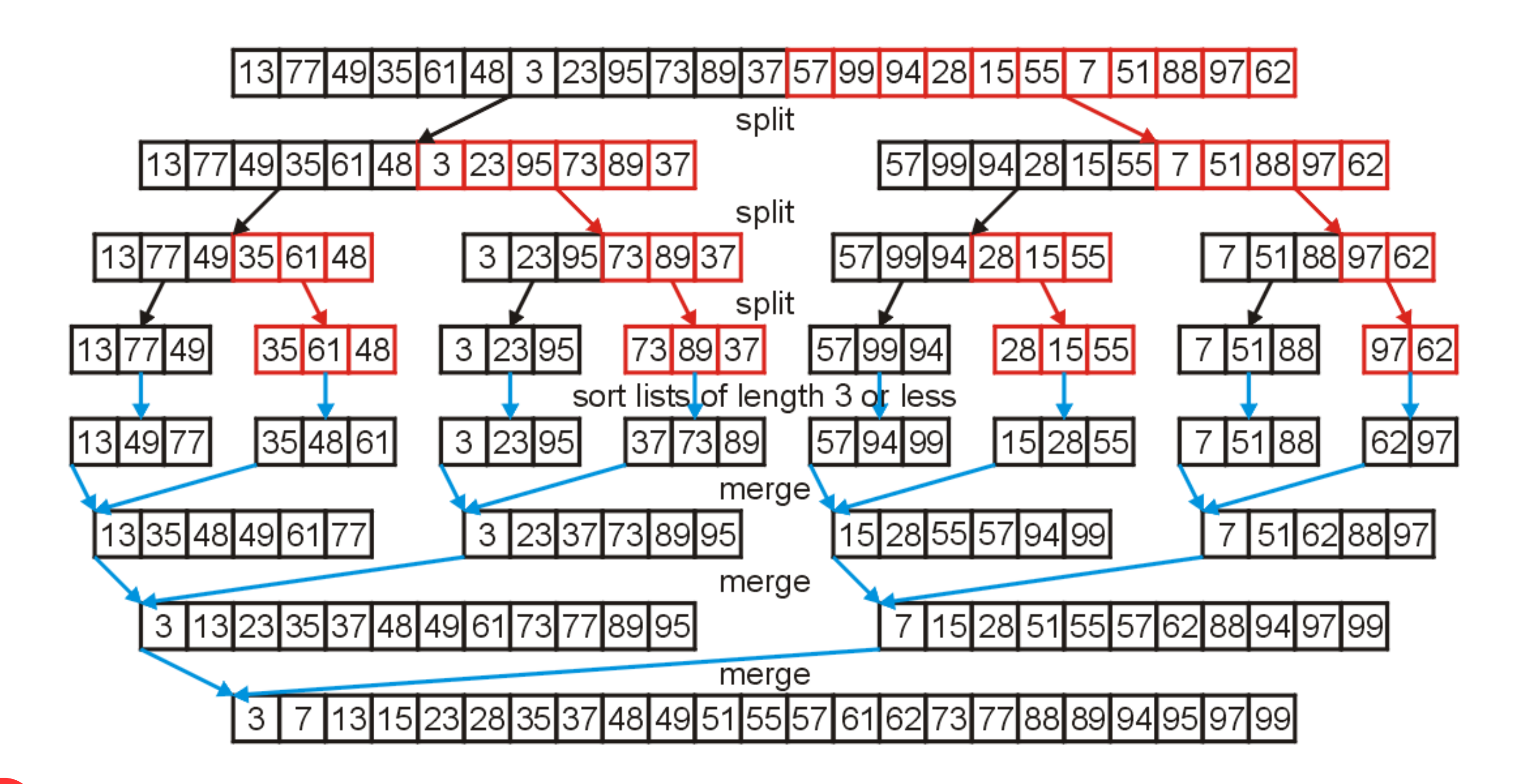
- Consideremos la siguiente lista de números desordenados

13 77 49 35 61 48 3 23 95 73 89 37 57 99 94 28 15 55 7 51 88 97 62

- Y ordenemos la misma usando merge sort

Merge Sort

- Aplicando el algoritmo tenemos:



Quick Sort

- Ya estudiamos un algoritmo de ordenación cuyo tiempo es $\mathbf{O}(n \log(n))$:
  - Merge sort, el cual es más rápido que los anteriores pero requiere más memoria adicional.
- Ahora conoceremos al Quick Sort.

Quick Sort

- Propuesto por C.A. Hoare en 1962
- Costo del Quicksort:
  - Este algoritmo, tiene un costo promedio de $\mathbf{O}(n \log(n))$, sin embargo el peor caso es $\mathbf{O}(n^2)$
- Existen modificaciones al algoritmo para evitar este peor caso.

Quick Sort

- El Mergesort trabaja partiendo el problema en 2 subproblemas y resolviendo cada subproblema.
- El problema más grande es partido en 2 subproblemas basados en la posición en el array.
- Consideremos la siguiente alternativa:
  - Elegir un elemento del array y partir los elementos restantes en 2 grupos relativos al elemento seleccionado.

Quick Sort

- Por ejemplo, dado

80 38 95 84 99 10 79 44 26 87 96 12 43 81 3

Podemos seleccionar al elemento 44, y ordenar los elementos restantes en 2 grupos: los menores que 44 y los mayores a 44:

38 10 26 12 43 3 44 80 95 84 99 79 87 96 81

- Si ordenamos cada sublista, entonces ya tendremos ordenado el array completo.

Quick Sort

- Así como en el mergesort, podemos:
  - Aplicar insertion sort si el tamaño de las sublistas es suficientemente pequeño, o
  - Ordenamos las sublistas utilizando usando quick sort

Quick Sort

- Si asumimos que la selección del pivote resulta en 2 particiones iguales (o aproximadamente iguales), entonces podríamos usar el mismo análisis usado para el mergesort
- Desafortunadamente, que pasa si siempre elegimos el menor elemento como pivote?

Mediana de Tres

- La elección ideal del pivote consiste en elegir el ***elemento mediano*** de la lista.
- Este elemento es difícil de encontrar, entonces
  - Elegimos la mediana entre los elementos que están al inicio, mitad y final de la lista

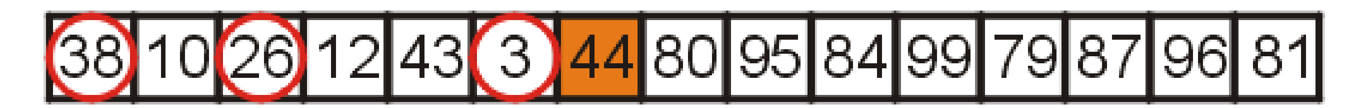


- Esto usualmente nos da una buena aproximación a la mediana actual de la lista

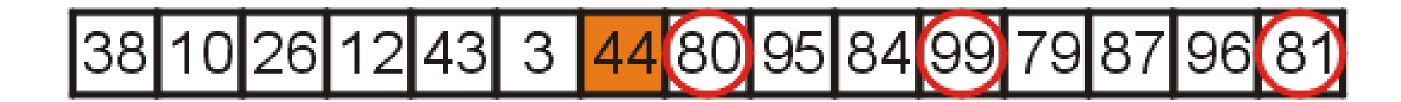


Mediana de Tres

- Ordenar los elementos basados en el 44, resulta en 2 sublistas, cada una de las cuales debe ser ordenada de nuevo usando quicksort.
- Seleccionamos el 26 para partir la primera sublista



Y 81 para partir la segunda sublista:



Mediana de Tres

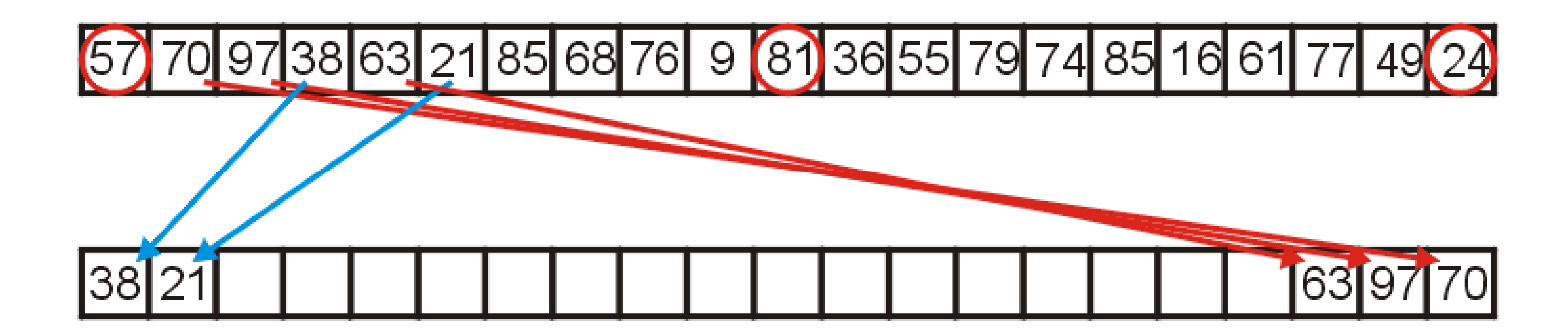
- La selección del pivote usando la mediana de tres, ayuda a acelerar el algoritmo.
- Una mala elección del pivote nos lleva al peor caso cuadrático.

Implementación

- Si elegimos utilizar memoria adicional para un segundo array, podemos implementar el particionamiento de la lista copiando elementos hacia el inicio o final del array auxiliar
- Finalmente, ubicamos al pivote en la posición vacía resultante.

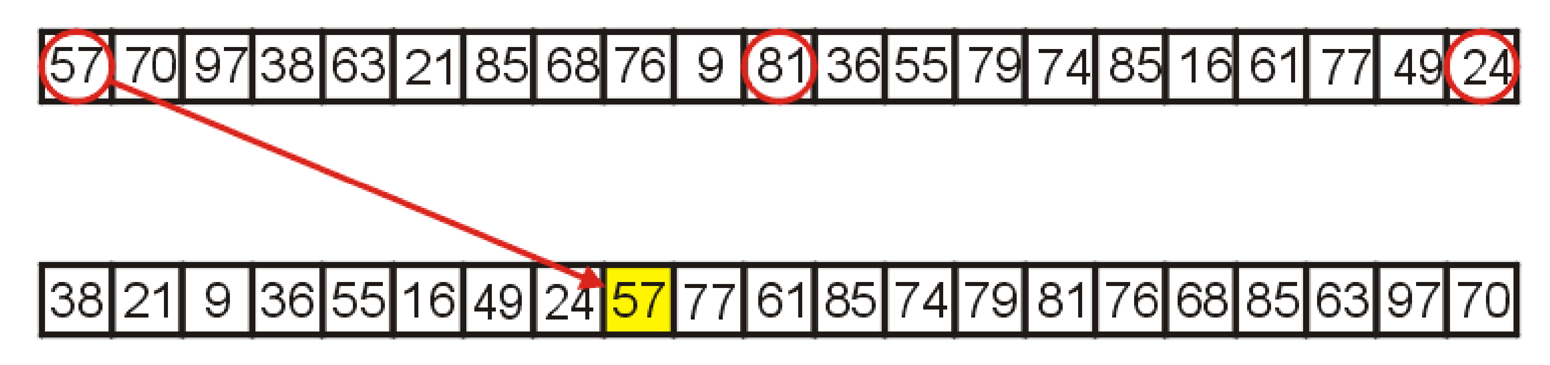
Implementación

- Por ejemplo, consideremos lo siguiente:
  - 57 es la mediana de tres
  - Recorremos los elementos restantes, asignandolos al inicio o fin de la lista del segundo array



Implementación

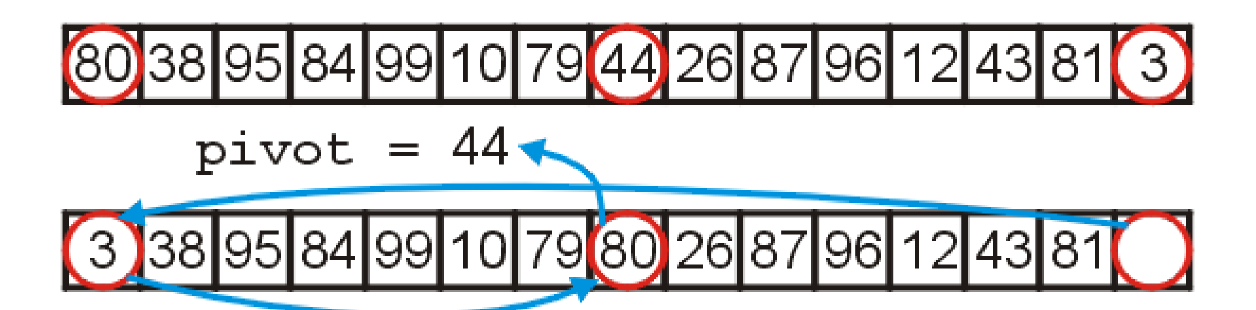
- Una vez finalizado, copiamos el pivote en la posición vacía resultante.



- ¿Podemos usar el quicksort sin un array adicional?

Implementación

- Al buscar el pivote por la mediana de tres, ya examinamos el primer, medio y último elemento de la lista.
- Adicionalmente, podríamos:
  - Mover al elemento más pequeño en la primera posición
  - Mover el elemento más grande en la mitad de la lista



Implementación

- Y luego, podemos encontrarnos con 2 tipos de elementos:
  - Mayores que el pivote
  - Menores que el pivote

  Los cuales están desordenados

Implementación

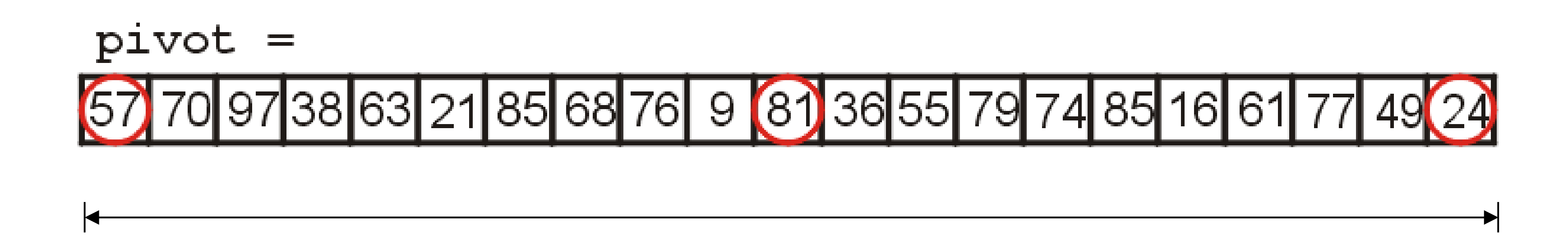
- Entonces, buscamos hacia adelante en la lista hasta encontrar un elemento mayor que el pivote y luego,
- Buscamos hacia atrás desde el final de la lista hasta encontrar un elemento menor que el pivote.
- Intercambiamos estas 2 entradas

Implementación

- Podemos repetir este proceso hasta que los índices se crucen y ahí se detiene la búsqueda
- El índice que apunta el mayor elemento después del cruce, puede utilizarse para mover al elemento almacenado allí a la última posición de la lista
- Y luego, en la posición donde estaba el mayor elemento referenciado por el índice, podemos ubicar al pivote.
- Así, el pivote ya quedará en su ubicación correcta en el array

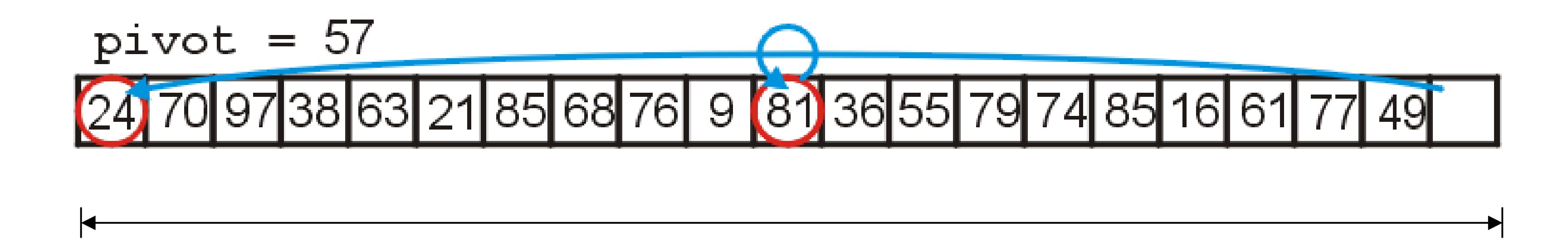
Ejemplo del Quick Sort

- Examinemos el concepto con un ejemplo concreto. Consideremos la lista de abajo
- Primero, examinamos los elementos ubicados al inicio, final y mitad de la lista
- La línea de abajo del array de ejemplo indica cuál lista está siendo ordenada.



Ejemplo del Quick Sort

- Seleccionamos el 57 como pivote.
- Movemos el 24 en la primera posición.
- El 81 permanece en su lugar en este caso.



Ejemplo del Quick Sort

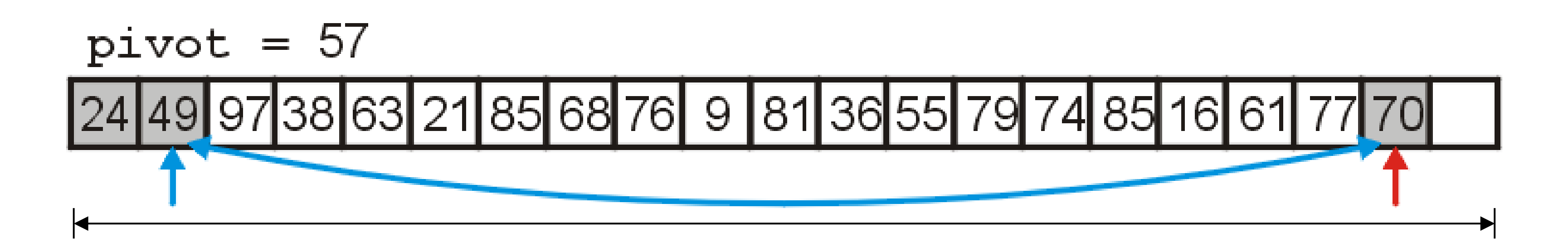
- Comenzando desde la 2ª y 2ª-última posiciones:
  - Buscamos hacia adelante hasta que encontramos 70 > 57
  - Buscamos hacia atrás hasta que encontramos 49 < 57

pivot = 57

24	70	97	38	63	21	85	68	76	9	81	36	55	79	74	85	16	61	77	49	

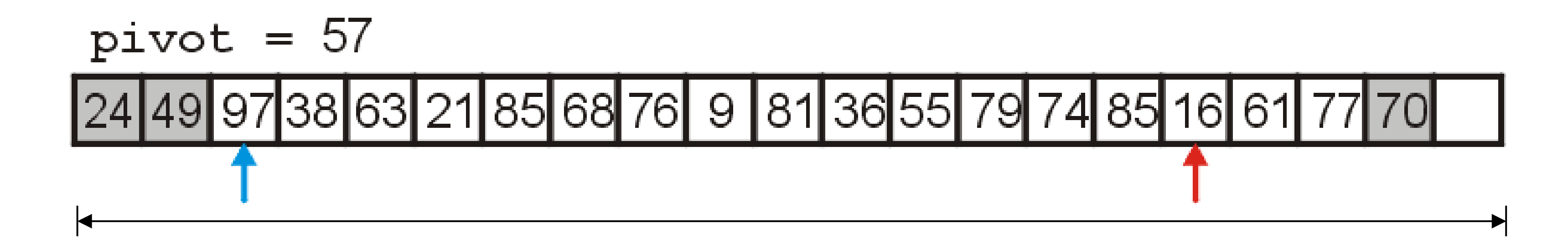
Ejemplo del Quick Sort

- Intercambiamos 70 y 49, ubicándolos en orden



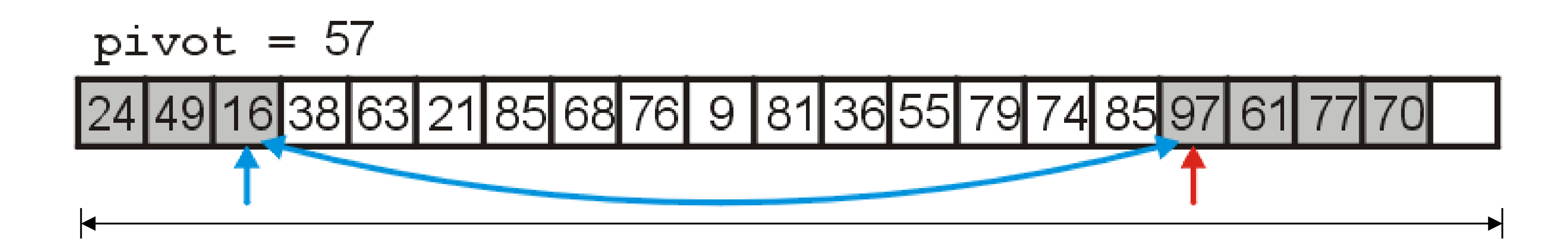
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 97 > 57
- Buscamos hacia atrás hasta 16 < 57



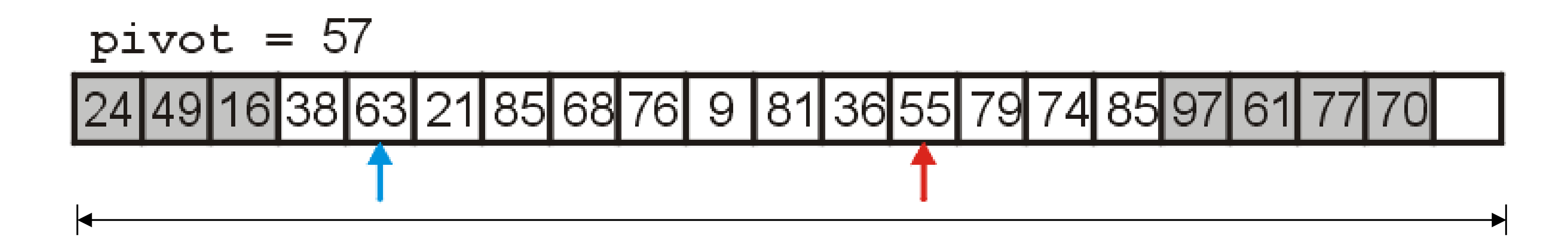
Ejemplo del Quick Sort

- Intercambiamos 16 y 97



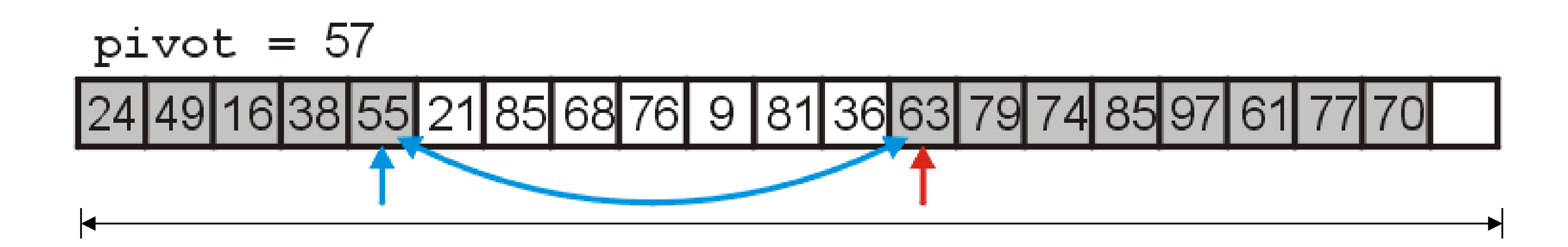
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 63 > 57
- Buscamos hacia atrás hasta 55 < 57



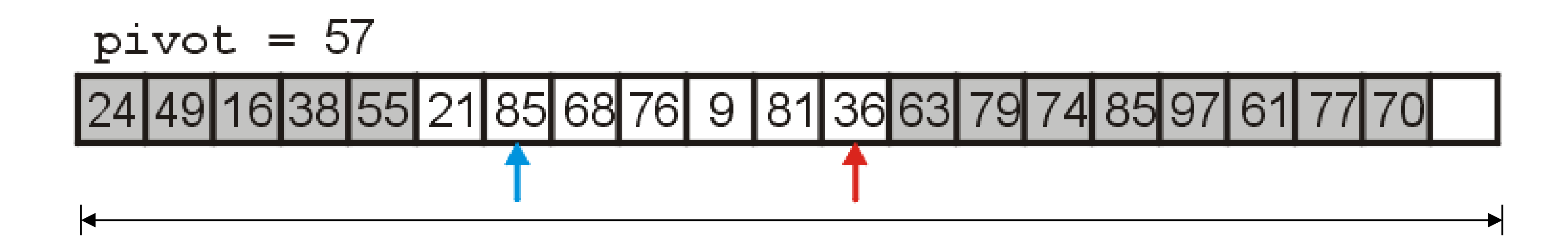
Ejemplo del Quick Sort

- Intercambiamos 63 y 55



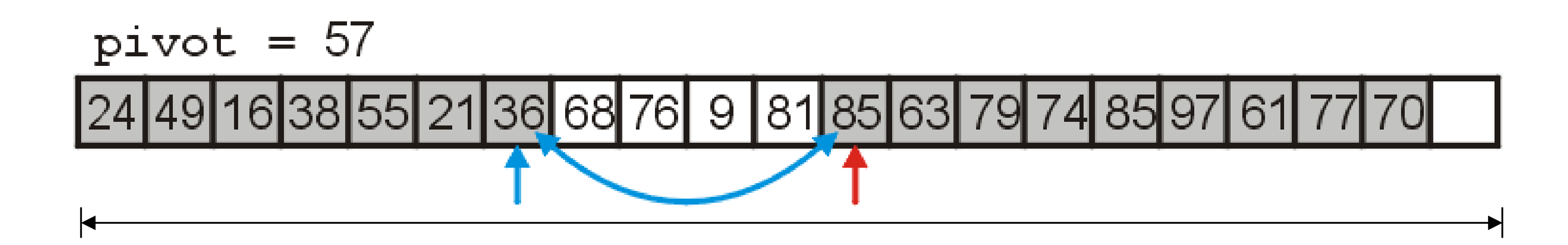
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 85 > 57
- Buscamos hacia atrás hasta 36 < 57



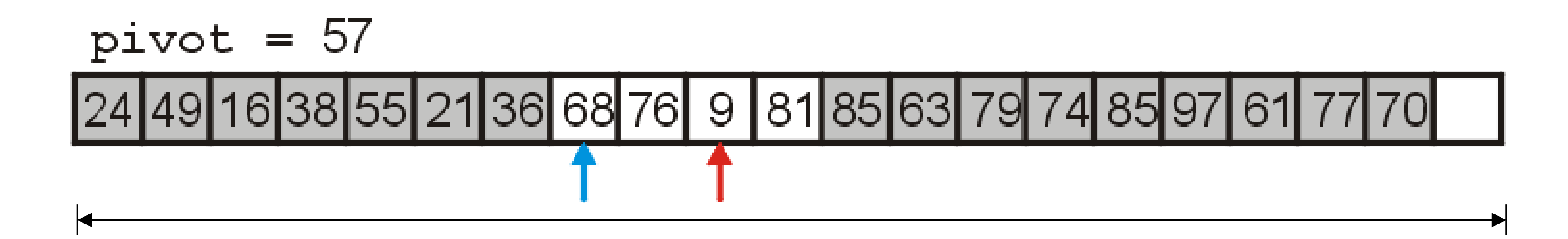
Ejemplo del Quick Sort

- Intercambiamos 85 y 36, ubicándolos en orden uno respecto al otro



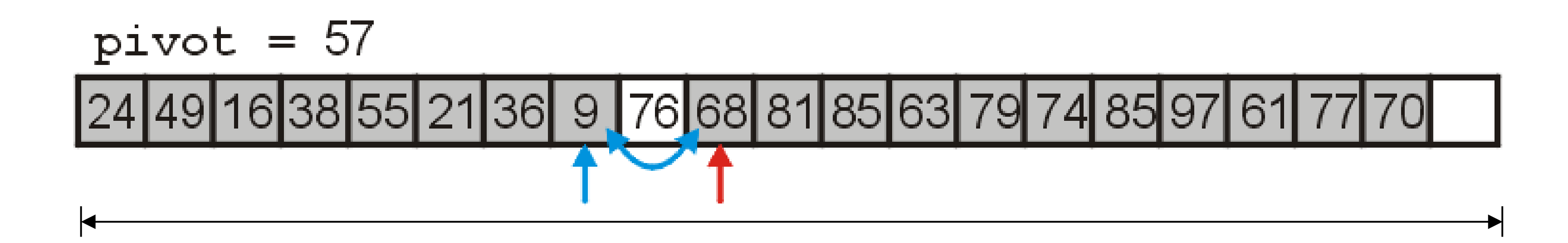
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 68 > 57
- Buscamos hacia atrás hasta 9 < 57



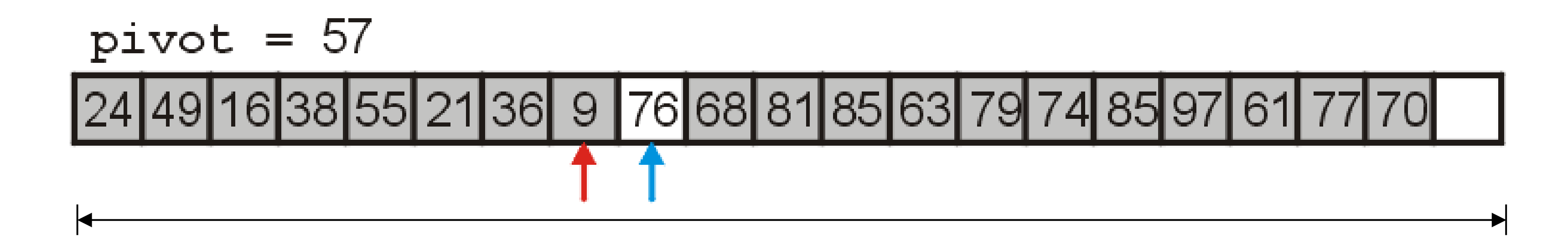
Ejemplo del Quick Sort

- Intercambiamos 68 y 9



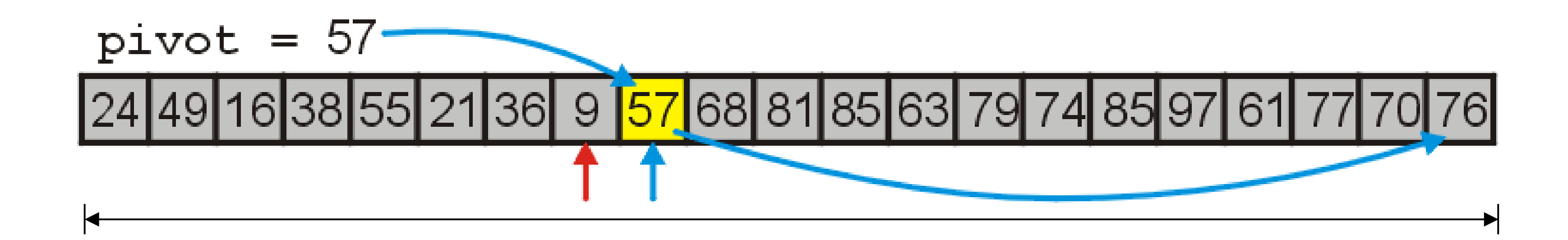
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta encontrar 76 > 57
- Buscamos hacia atrás hasta encontrar 9 < 57
- Los índices están cruzados (fuera de orden) así que paramos



Ejemplo del Quick Sort

- Movemos el mayor elemento indexado a la posición vacía al final del array
- La nueva posición vacía es completada son el pivote, 57.
- El pivote ahora está en su ubicación correcta.



Ejemplo del Quick Sort

- Ahora, recursivamente llamamos a quick sort para que ordene la primera mitad de la lista
- Cuando termine, todos los elementos < 57 quedarán ordenados.



Ejemplo del Quick Sort

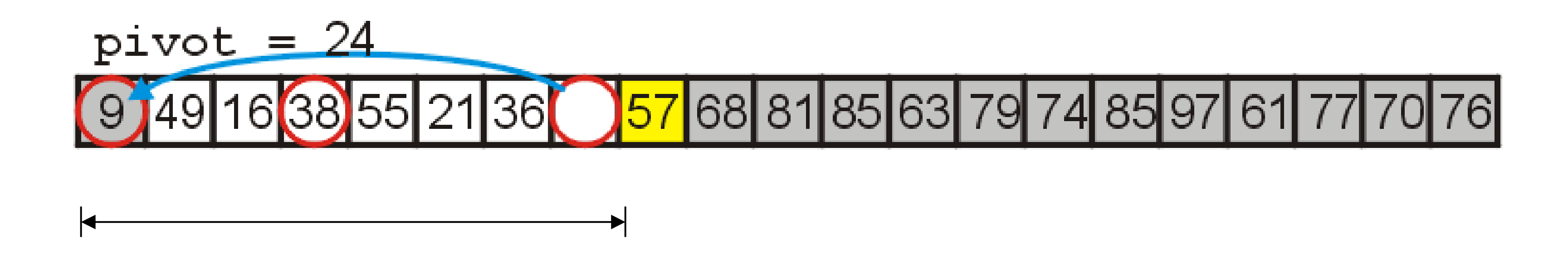
- Examinamos los elementos ubicados en las posiciones: primero, último y mitad.

pivot =

24	49	16	38	55	21	36	9	57	68	81	85	63	79	74	85	97	61	77	70	76

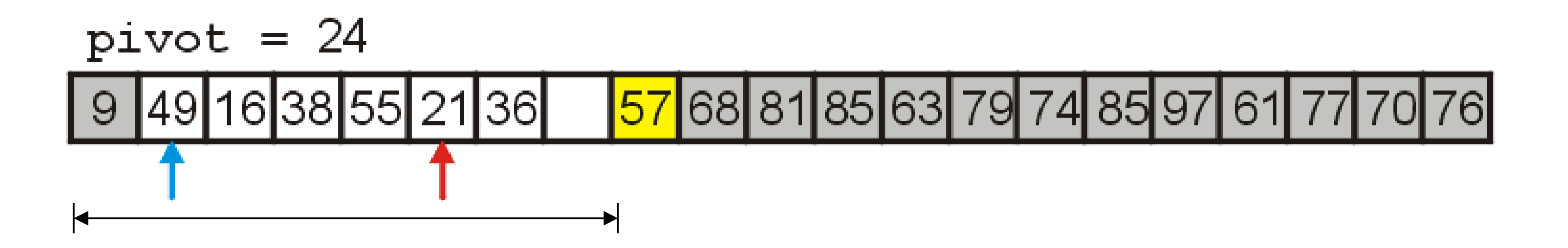
Ejemplo del Quick Sort

- Elegimos 24 como pivote.
- Movemos 9 en la primera posición de la sublista.



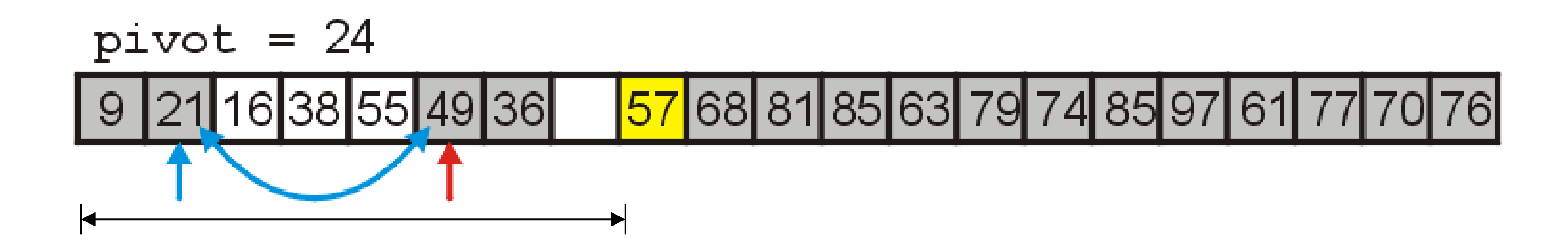
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 49 > 24
- Buscamos hacia atrás hasta 21 < 24



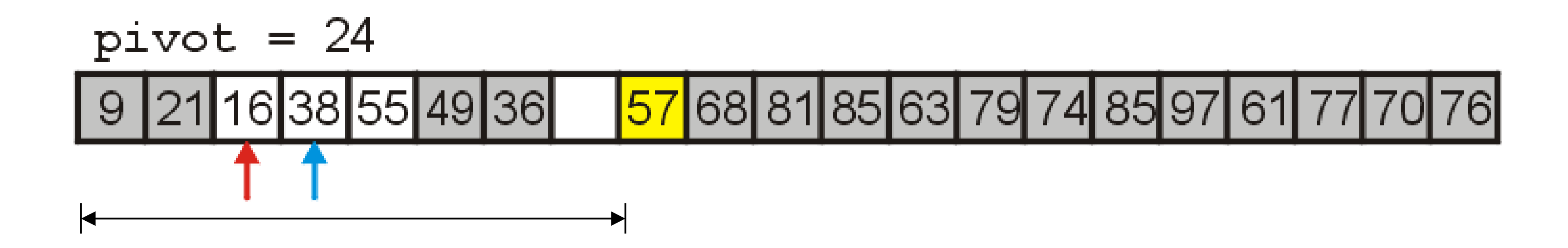
Ejemplo del Quick Sort

- Intercambiamos 49 y 21



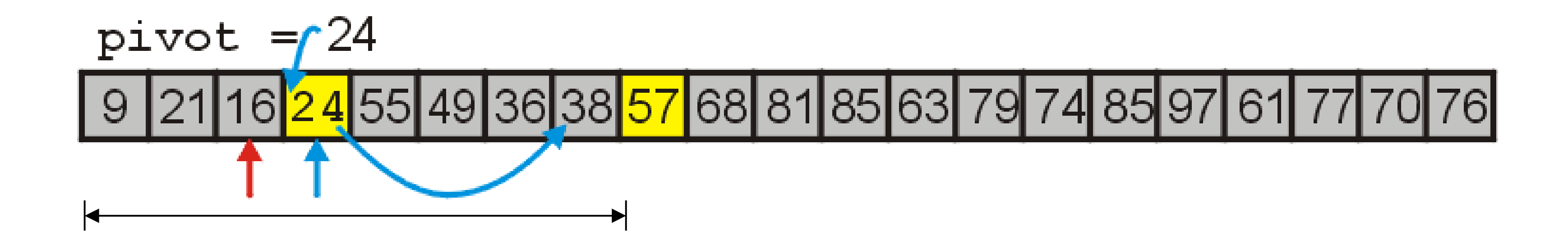
Ejemplo del Quick Sort

- Buscamos hacia adelante hasta 38 > 24
- Buscamos hacia atrás hasta 16 < 24
- Los índices están cruzados, así que paramos.



Ejemplo del Quick Sort

- Movemos el 38 a la posición vacía al final de la sublista y movemos el pivote 24 en la posición donde previamente estaba el 38
- El elemento 24 está ahora en su ubicación correcta.



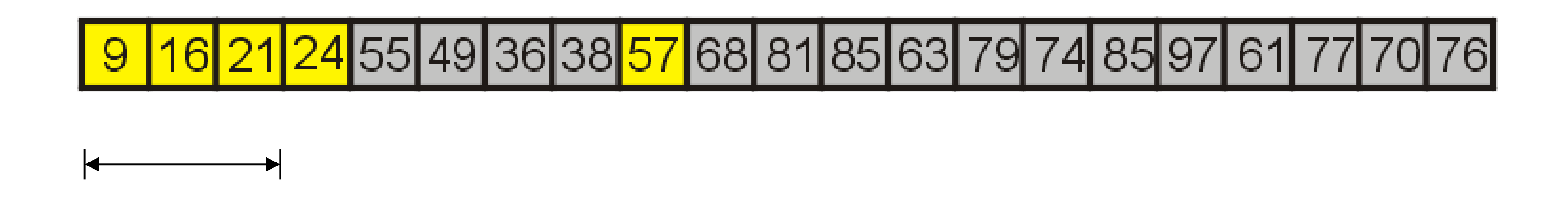
Ejemplo del Quick Sort

- Recursivamente llamaremos al quick sort para ordenar las sublistas izquierda y derecha, de los elementos que son menores que 57

9	21	16	24	55	49	36	38	57	68	81	85	63	79	74	85	97	61	77	70	76

Ejemplo del Quick Sort

- La primera partición tiene 3 entradas, las ordenamos usando insertion sort



Ejemplo del Quick Sort

- La segunda partición tiene solo 4 entradas, así que también las ordenamos usando insertion sort.
- Luego ordenamos la lista derecha ubicada luego del elemento 57.



Ejemplo del Quick Sort

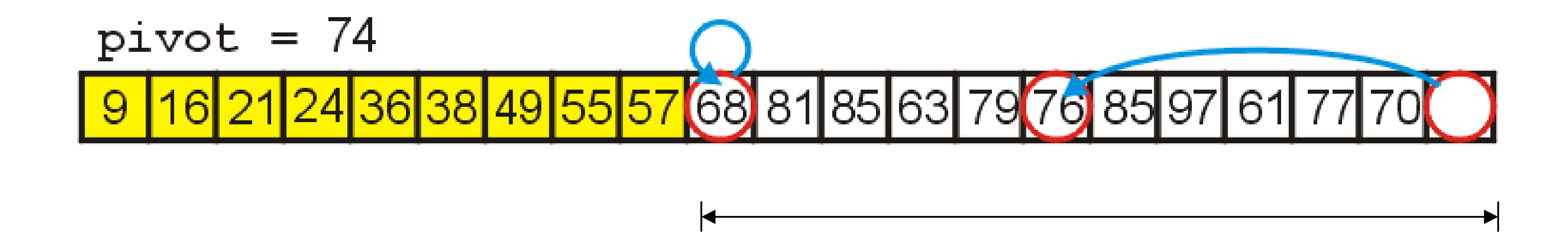
- Examinamos elementos buscando el pivote

pivot =

9	16	21	24	36	38	49	55	57	68	81	85	63	79	74	85	97	61	77	70	76

Ejemplo del Quick Sort

- Elegimos al 74 como pivote
- Movemos al 76 a la posición donde estaba el 74



Ejemplo del Quick Sort

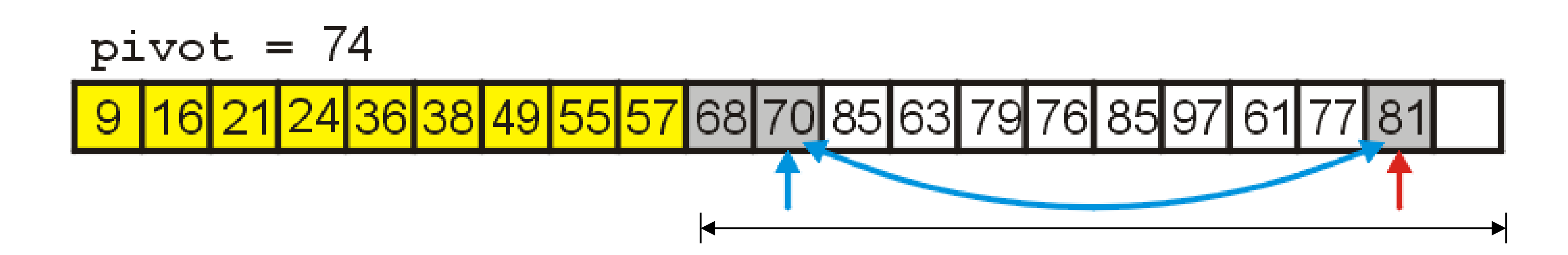
- Buscamos hacia adelante hasta 81 > 74
- Buscamos hacia atrás hasta 70 < 74

pivot = 74

9	16	21	24	36	38	49	55	57	68	81	85	63	79	76	85	97	61	77	70	

Ejemplo del Quick Sort

- Intercambiamos 70 y 81



Ejemplo del Quick Sort

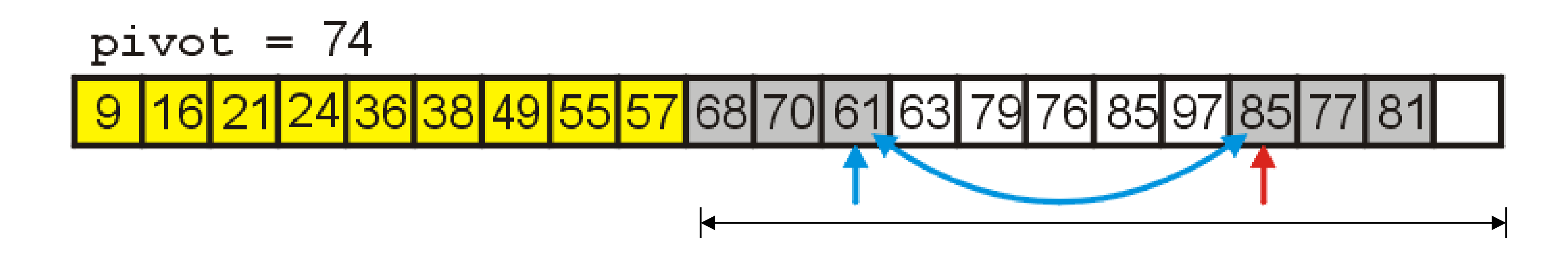
- Buscamos hacia adelante hasta 85 > 74
- Buscamos hacia atrás hasta 61 < 74

pivot = 74

9	16	21	24	36	38	49	55	57	68	70	85	63	79	76	85	97	61	77	81	

Ejemplo del Quick Sort

- Intercambiamos 85 y 61



Ejemplo del Quick Sort

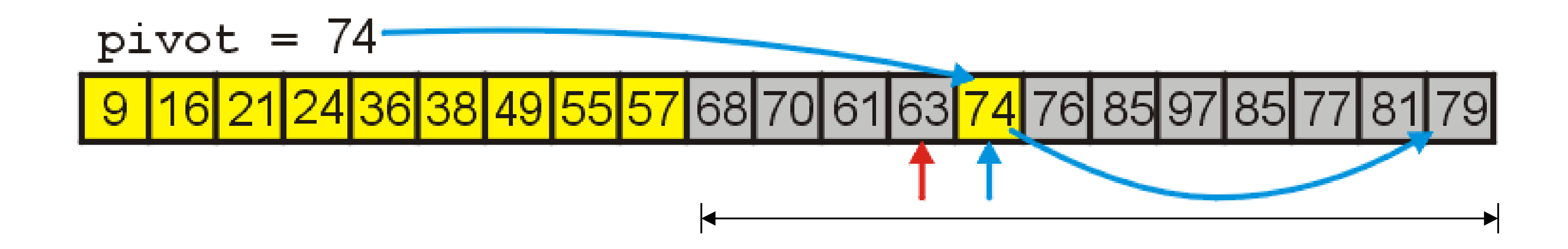
- Buscamos hacia adelante hasta 79 > 74
- Buscamos hacia atrás hasta 63 < 74
- Los índices se han cruzado, así que paramos.

pivot = 74

9	16	21	24	36	38	49	55	57	68	70	61	63	79	76	85	97	85	77	81	

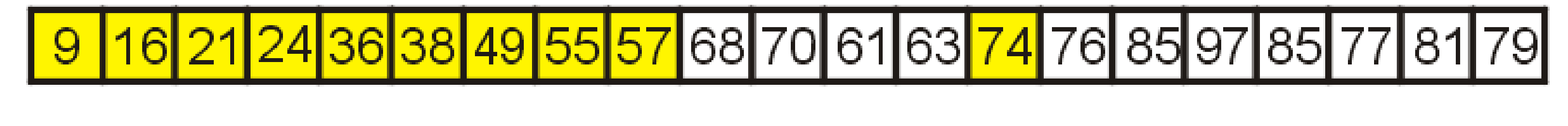
Ejemplo del Quick Sort

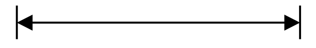
- Movemos al 79 en la posición vacía al final de la lista y luego movemos al pivote 74 en la posición ocupada previamente por el 79
- El 74 está ahora en su ubicación correcta.



Ejemplo del Quick Sort

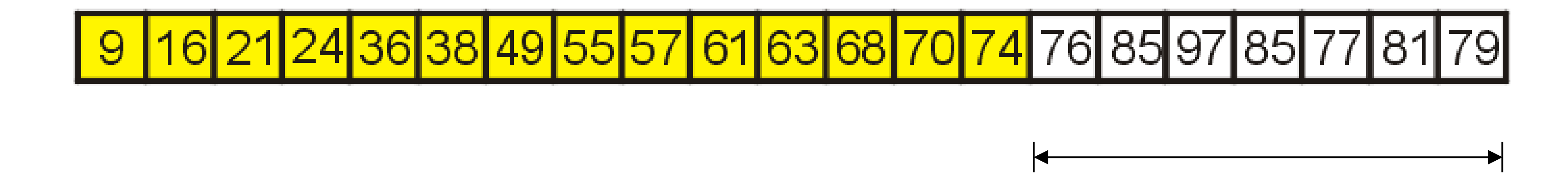
- Ordenamos la sublista izquierda con insertion sort porque sólo tiene 4 elementos





Ejemplo del Quick Sort

- Luego ordenamos la sublista derecha.



Ejemplo del Quick Sort

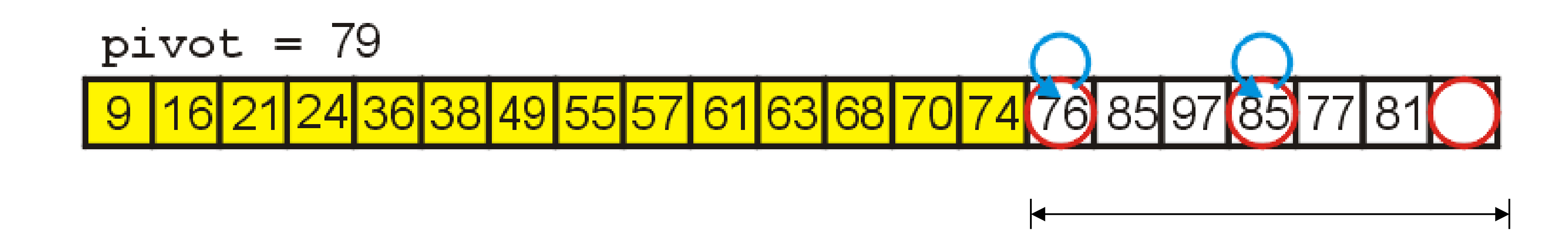
- Examinamos elementos para seleccionar al pivote

pivot =

9	16	21	24	36	38	49	55	57	61	63	68	70	74	76	85	97	85	77	81	79

Ejemplo del Quick Sort

- Elegimos al 79 como pivote y movemos:
  - 76 a la menor posición
  - 85 en la mayor posición



Ejemplo del Quick Sort

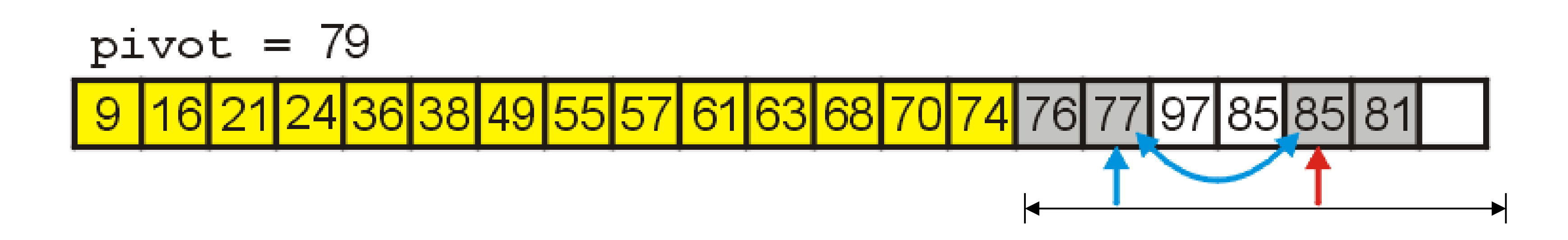
- Buscamos hacia adelante hasta 85 > 79
- Buscamos hacia atrás hasta 77 < 79

```
pivot = 79
```

9	16	21	24	36	38	49	55	57	61	63	68	70	74	76	85	97	85	77	81	

Ejemplo del Quick Sort

- Intercambiamos 85 y 77



Ejemplo del Quick Sort

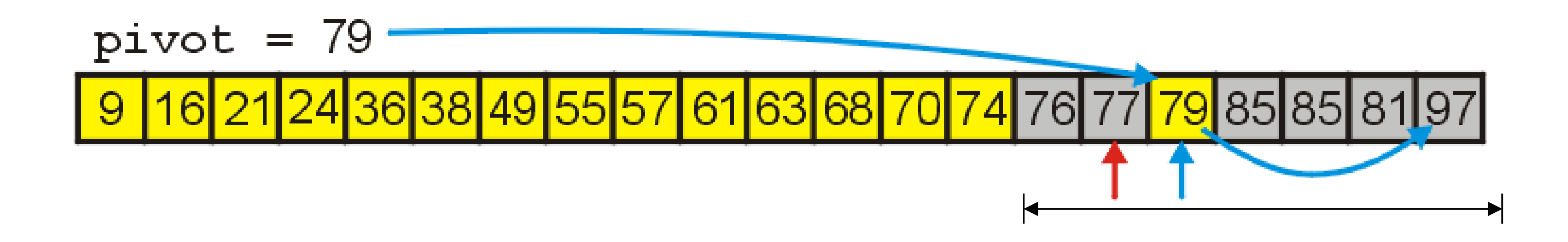
- Buscamos hacia adelante hasta 97 > 79
- Buscamos hacia atrás hasta 77 < 79
- Los índices están invertidos así que paramos.

pivot = 79

9	16	21	24	36	38	49	55	57	61	63	68	70	74	76	77	97	85	85	81	

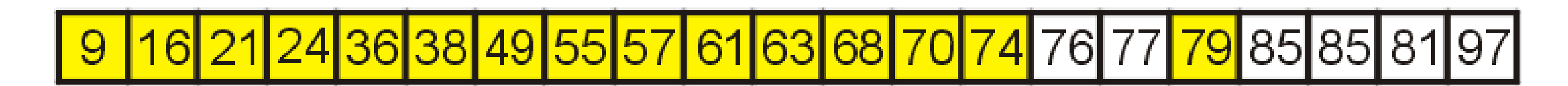
Ejemplo del Quick Sort

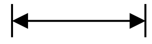
- Finalmente 97 va a la posición vacía al final de la lista y el 79 se ubica en su posición correcta.



Ejemplo del Quick Sort

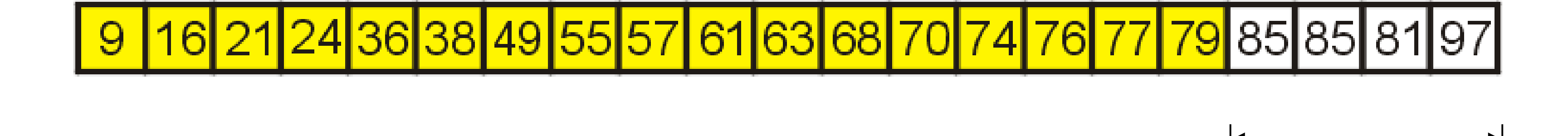
- Esto genera 2 sublistas de tamaño 2 y 4 respectivamente.
- Usamos insertion sort en la primera sublista.





Ejemplo del Quick Sort

- Usamos insertion sort en la segunda sublista, porque tiene solo 4 entradas



Ejemplo del Quick Sort

- Al ordenar la última sublista, tenemos la lista completa ordenada.

9	16	21	24	36	38	49	55	57	61	63	68	70	74	76	77	79	81	85	85	97

Otros algoritmos de ordenación?

- En cuanto a los algoritmos de ordenación estudiados, hay otros más dispersos por el mundo.. bastante eficientes y famosos, algunos inclusive lineales en ciertos contextos...
  - HeapSort
  - BucketSort
  - RadixSort
  - BinSort
  - BrickSort
  - CombSort
  - … entre otros ...

Referencias

- Estructuras de Datos en Java. Mark Allen Weiss, Capítulos 1, 8 y 20.
- A Practical Introduction to Data Structures and Algorithm Analysis. Shaffer, Cap. 8
- Algoritmos y Estructuras de Datos. Alfred Aho. Cap. 8
- Data Structures and Algorithm Analysis. *Mark Allen Weiss*
- ECE 250 Data Structures and Algorithms, University of Waterloo. *Douglas Wilhelm Harder*

Gracias por su Atención

¿Consultas?